CHAPTER 16

BUT I DON'T DO WINDOWS!

Most of this book has been describing examples using the Windows Driver Model and Win32 API system calls. This, of course, binds them to a Windows 98 or Windows 2000 PC host platform. I received many emails from readers of my first book requesting that I include other operating systems examples.

The two most requested operating systems were Linux and "a real time operating system" – I chose VxWorks from Wind River Systems. This chapter looks inside each of these operating systems with a special focus on their USB capabilities. The "Buttons and Lights" example that we worked in Chapter 7 is used as design target so that you can compare the solutions. I must thank Vojtech Pavlik of SuSE Labs, and Ted Hartnell and Srinagesh Modereti of Wind River Systems for their contributions to this chapter.

THE LINUX KERNEL

Linux was the brainchild of Linus Torvalds who, while an undergraduate at the University of Helsinki in 1991, motivated a group of talented programmers to create a new version of UNIX for an Intel 386 PC platform. One of the most distinguishing features that Linus insisted upon was that all of the kernel source code should be freely distributable. Linus, and other contributors, still hold the copyright and, to this day, the kernel sources are still freely distributable under the terms of the GNU General Public License (a copy is on the CDROM in the Chapter_16/Linux directory).

Linux, per se, is only the kernel of the operating system, the part that controls hardware, manages files, separates processes, and so on. There are several combinations of Linux with sets of utilities and applications added to form a complete operating system. Each of these combinations is called a distribution of Linux. The word Linux, though, in its strictest form, refers specifically to the kernel. It is also widely and incorrectly used to refer to an entire operating system built around the Linux kernel.

Major distributions are SuSE, Red Hat, Caldera OpenLinux, Mandrake and TurboLinux. None of these distributions is “the official Linux”.

Linux runs on any 32-bit Intel processor, ranging from a 386SX to Pentium-4, from embedded machines to large superservers. Linux has also been ported and to other architectures.

Linux is not public domain, nor is it “shareware”. It is “free” software, commonly called Open Source Software[tm] (see http://www.opensource.org), or freeware, and you may give away or sell copies, but you must include the source code or make it available in the same way as any binaries you give or sell. If you distribute any modifications, you are legally bound to distribute the source for those modifications. Note carefully that the “free” part involves access to the source code rather than money; it is perfectly legal to charge money for distributing Linux, so long as you also distribute the source code. This is a generalization; if you need finer detail, please read the GPL.

Linux is developed using an open and distributed model, rather than a closed and centralized model like most other software. This means that the current development version is always public (with up to a week or two of delay) so that anybody can use it. The result is that whenever a version with new functionality is released, it almost always contains bugs, but it also results in a very rapid development so that the bugs are found and corrected quickly, often in hours, as many people work to fix them. In contrast, the closed and centralized model means that there is only one person or team working on the project, and they only release software that they think is working well. Often this leads to long intervals between releases, long waiting for bug fixes, and slower development. The latest release of such software to the public is sometimes of higher quality, but the development speed is generally much slower.

How critical is a timely correction? Well, in 1997, a problem with TCP/IP resulting from a deliberate misuse of a program called PING affected almost every UNIX system in the world. This problem would allow a hacker to shut down other people’s UNIX systems. The world needed a fix – and quickly! A fix from Linux was on the Internet just four hours after the problem was identified, and people could get the fix easily and apply it to their systems. Various commercial UNIX vendors took weeks to develop the fix and send it to their customers.

Linux has an impressive collection of high level features:

Multitasking: several programs running at the same time.

Multi-user: several users on the same machine at the same time

Multiprocessor: runs on computers with SMP architecture

Multithreading: has native kernel support for multiple independent threads of control within a single process memory space.

Memory protection between processes: so that one program can't bring the whole system down.

Demand loads executables: Linux only reads from disk those parts of a program that are actually used.

Linux is also a very deep and broad implementation with the list of secondary features going on for several pages (see the CDROM).

Linux is a fully-fledged server and workstation operating system, capable of running internet and intranet services, web, database, filesharing applications, also equipped with a user friendly graphical interface. User applications for it include word processor, spreadsheets, databases, web and e-mail clients, simply anything anyone can want for a desktop computer.

USB FEATURES

The Linux USB stack was started in 1998 by Linux Torvalds himself. A team formed for rapid development of the USB drivers and today Linux has almost complete support for USB class devices as well as drivers for many vendor specific interfaces, ranging from USB CDC ACM modems to SmartMedia USB card readers.

The Linux USB stack consists of a core driver (usb-core.o), host controller drivers (uhci-hcd.o, ohci-hcd.o) and device drivers (acm.o, printer.o, hid.o ...). These drivers are Linux kernel modules that load and unload during runtime. This implementation speeds up development rapidly, because there is no need to reboot when testing a new development version of a driver. The source code for all of these USB elements is included on the CDROM in the Chapter_16/Linux directory.

Buttons and Lights Example

One of the attributes we have learnt about the EZ-USB component is that it is soft loadable. When an I/O device containing an EZ-USB component (i.e. just like our hardware) is attached to a PC host then some process on that PC host must download firmware into the component. The Linux community have developed such a loader and its source is also on the companion CDROM (it's called dl).

A free 8051 cross-assembler is available from http://sdcc.sourceforge.net (actually a cross C-compiler is also available for free download from this site). The original firmware source code developed in Chapter 7 was re-assembled and downloaded to the "button and Lights" hardware – it enumerated as a standard HID device. Linux contains all of the code to handle a HID device so no OS software or USB driver software had to be written. The application code required to exercise the buttons and lights hardware is shown in Figure 16-1. It is a simple C program that opens a Linux input device and handles events.

```
/*
 * $Id: leds.c,v 1.1 2000/08/17 18:56:37 vojtech Exp $
 *
 *  Copyright (c) 1999-2000 Vojtech Pavlik
 *
 *  Leds'n'buttons test program
 */

/*
 * This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * Should you need to contact me, the author, you can do so either by
 * e-mail - mail your message to <vojtech@ucw.cz>, or by paper mail:
 * Vojtech Pavlik, Ucitelska 1576, Prague 8, 182 00 Czech Republic
 */

#include <linux/input.h>

#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main (int argc, char **argv)
```

```
{
        int fd, rd, i;
        struct input_event ev[64];
        int version;
        unsigned short id[4];
        char name[256] = "Unknown";

        if (argc < 2) {
                printf ("Usage: leds /dev/inputX\n");
                printf ("Where X = input device number\n");
                exit (1);
        }
        if ((fd = open(argv[argc - 1], O_RDWR)) < 0) {
                perror("leds");
                exit(1);
        }
        ioctl(fd, EVIOCGVERSION, &version);

        printf("Input driver version is %d.%d.%d\n",
                version >> 16, (version >> 8) & 0xff, version & 0xff);

        ioctl(fd, EVIOCGID, id);

        printf("Input device ID: bus 0x%x vendor 0x%x product 0x%x version 0x%x\n",
                id[ID_BUS], id[ID_VENDOR], id[ID_PRODUCT], id[ID_VERSION]);

        ioctl(fd, EVIOCGNAME(sizeof(name)), name);

        printf("Input device name: \"%s\"\n", name);

        if (id[ID_BUS] != BUS_USB || id[ID_VENDOR] != 0x4242 || id[ID_PRODUCT] != 0x4201) {
                fprintf(stderr, "leds: %s is not a Leds & Buttons device\n", argv[argc - 1]);
                return -1;
        }

        printf("Testing ... (interrupt to exit)\n");

        while (1) {
                rd = read(fd, ev, sizeof(struct input_event) * 64);
                if (rd < sizeof(struct input_event)) {
                        perror("leds: error reading");
                        exit (1);
                }
                for (i = 0; i < rd / sizeof(struct input_event); i++) {
                        printf("%s %d: %s\n",
                                ev[i].type == EV_KEY ? "Button" : "LED   ",
                                ev[i].code - (ev[i].type == EV_KEY ? BTN_MISC : LED_MISC),
                                ev[i].value ? "on" : "off");
                        if (ev[i].type == EV_KEY) {
                                ev[i].type = EV_LED;
                                ev[i].code += LED_MISC - BTN_MISC;
                                write(fd, ev + i, sizeof(struct input_event));
                        }
                }
        }
}
```

Figure 16-1. PC host source code for Linux example

THE VXWORKS KERNEL

The PC platform has found its way into industrial control, automation and data acquisition systems. The form factor had to change for this non-office environment. Figure 16-2 shows an industrial PC from Radisys Corporation; it contains all of the components you would find on a desktop PC motherboard and it will therefore run all PC software. Some applications use a Windows operating system but the preferred OS for this environment is a real time operating system (RTOS) such as Wind River's VxWorks.

<< Picture on it's way from Radisys Corp. >>

Figure 16-2. An industrial design PC Platform

The operation of an RTOS is deterministic since it has guaranteed response times – this is REQUIRED when dealing with real world events like motors, heaters, temperature sensors and the whole realm of physical transducers. USB is a popular interconnect scheme in this application too since its hot plug-and-play capabilities makes it easy to configure test and measurement equipment. A variety of USB I/O devices targeted for this market segment are already available (Figure 16-3).

Courtsey of Hohner Corp.

Figure 16-3. Industrial Sensors using USB

An RTOS also has to be multi-threaded/multi-tasking and pre-emptible. The notion of thread priority has to exist and the OS has to support predictable thread synchronization mechanisms with priority inheritance.

Real-time Determinism

By definition, real-time applications are required to respond to external events within predictable time limits. This is especially true of "hard" real-time systems, (ie programs interfacing with the real world) where missed deadlines can have dire, or even disastrous, consequences.

A real-time system's ability to respond to external events within a specified time in known as *determinism*. To indicate how well an RTOS can support determinism, most vendors quote at least the following performance metrics (see Figure 16-4):

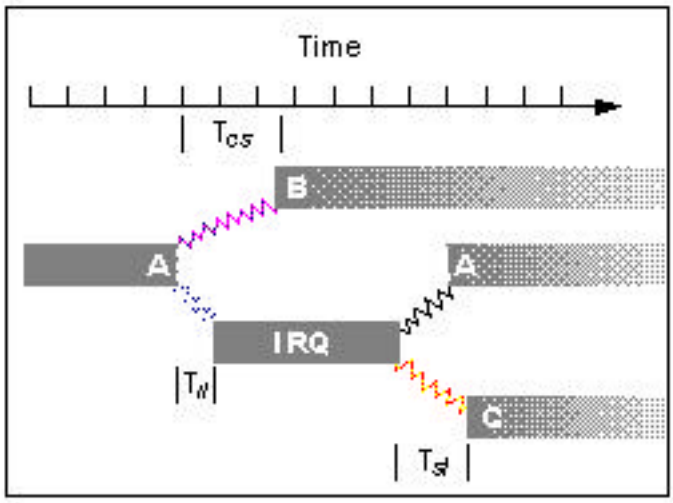

Figure 16-4 – Key metrics of an operating system's realtime determinism

interrupt latency: (til) the time from the start of the physical interrupt to the execution of the first instruction of the user-written interrupt service routine

scheduling latency (Interrupt Dispatch Latency): (tsl) the time from the execution of the last instruction of the user-written interrupt handler to the first instruction of the process made "ready" by that interrupt

context-switch time: (tcs) the time from the execution of the last instruction of one user-level process to the first instruction of the next user-level process

maximum system call time: should be predictable and independent from the number of objects in the system

A good RTOS is more than a good Kernel. A good RTOS should have a good documentation, should be delivered with good tools to develop and tune your application. Good values for the Interrupt latency and Context switch time are important, but there are a lot of other parameters that will make a good RTOS. For example a RTOS supporting many devices will have more advantages than a simple very good nano-kernel. The relationship of the VxWorks kernel and operating system is shown in Figure 16-5.

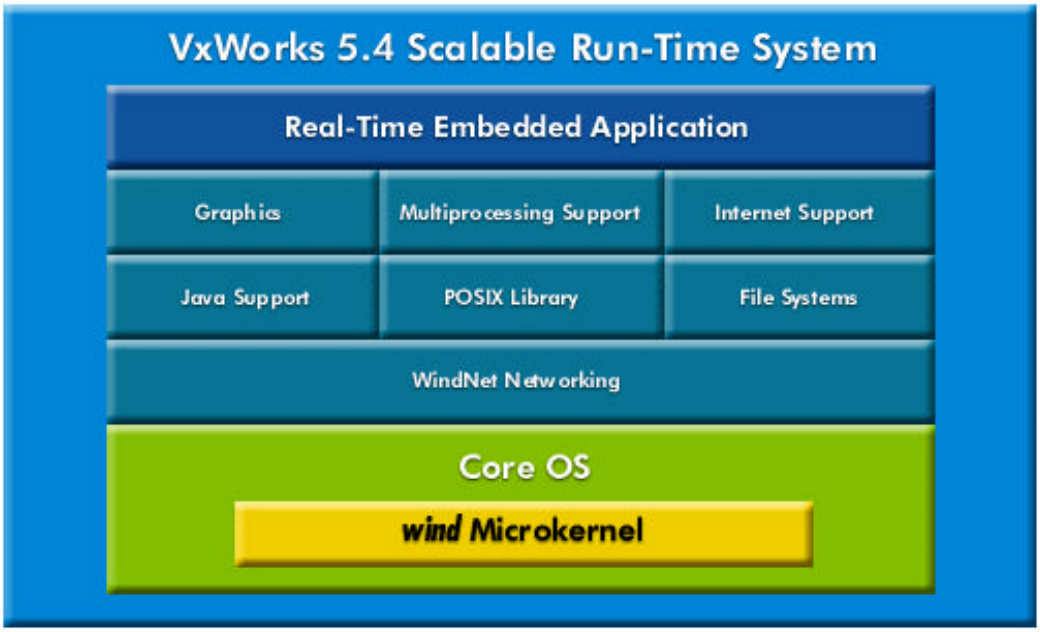

Figure 16-5. An operating system builds on its kernel

VxWorks, is the run-time component of the Tornado® II embedded development platform. Tornado II also includes a comprehensive suite of core and optional cross-development tools and utilities and a full range of communications options for the target connection to the host. An evaluation set of these tools is included on the CDROM.

VxWorks is flexible, with more than 1800 powerful application program interfaces (APIs); scaleable, from the simplest to the most complex product designs; reliable, used in mission-critical applications ranging from anti-lock braking systems to inter-planetary exploration.

The VxWorks RTOS comprises the core capabilities of the wind® microkernel along with advanced networking support, powerful file system and I/O management, and C++ and other standard run-time support. These core capabilities can be combined with add-on components available from Wind River Systems.

High-performance Microkernel Design

At the heart of the VxWorks run-time system is the wind microkernel. This microkernel supports a full range of real-time features including fast multitasking, interrupt support, and both preemptive and round robin scheduling. The microkernel design minimizes system overhead and enables fast, deterministic response to external events.

The run-time environment also provides efficient intertask communication mechanisms, permitting independent tasks to coordinate their actions within a real-time system. The developer may design applications using shared memory (for simple sharing of data), message queues and pipes (for intertask messaging within a CPU), sockets and remote procedure calls (for network-transparent communication), and signals (for exception handling). For controlling critical system resources, several types of semaphores are provided – binary, counting, and mutual exclusion with priority inheritance.

Scaleable run-time software

VxWorks is designed for scalebility, enabling developers to allocate scarce memory resources to their application, rather than to the operating system. From deeply embedded designs requiring a few kilobytes of memory, to complex high-end real-time systems where more operating system functions are needed, the developer may choose from over 100 different options to create hundreds of configurations. Individual modules may be used in development and omitted in production systems.

Furthermore, these individual subsystems are themselves scaleable, allowing the developer to optimally configure VxWorks run-time software for the widest range of applications. For example, individual functions may be removed from the ANSI C run-time library, or specific kernel synchronization objects may be omitted if they are not required by the application. Also, TCP, UDP, sockets, and standard Berkeley network services can all be scaled in or out of the networking stack as necessary.

These configuration options can be easily selected by means of the Tornado II project facility’s graphical interface. Developers can also use Tornado II´s autoscaling feature, which automatically analyzes application code and incorporates the appropriate options.

USB in VxWorks:

Wind River's USB Developer's Kit enables developers to provide standard universal serial bus (USB) connectivity within embedded products and attached peripheral devices. Although USB was originally designed to improve connectivity between a personal computer and its peripheral devices, demand has been growing for USB connectivity in embedded products used in industries such as digital imaging, industrial automation, medical, home networking, global communications, transportation, and test equipment. The USB Developer's Kit includes both host and peripheral driver stacks (Figure 16-6). Thus enabling developers to build USB host and peripheral products and create customized classes of drivers on a wide range of applications.

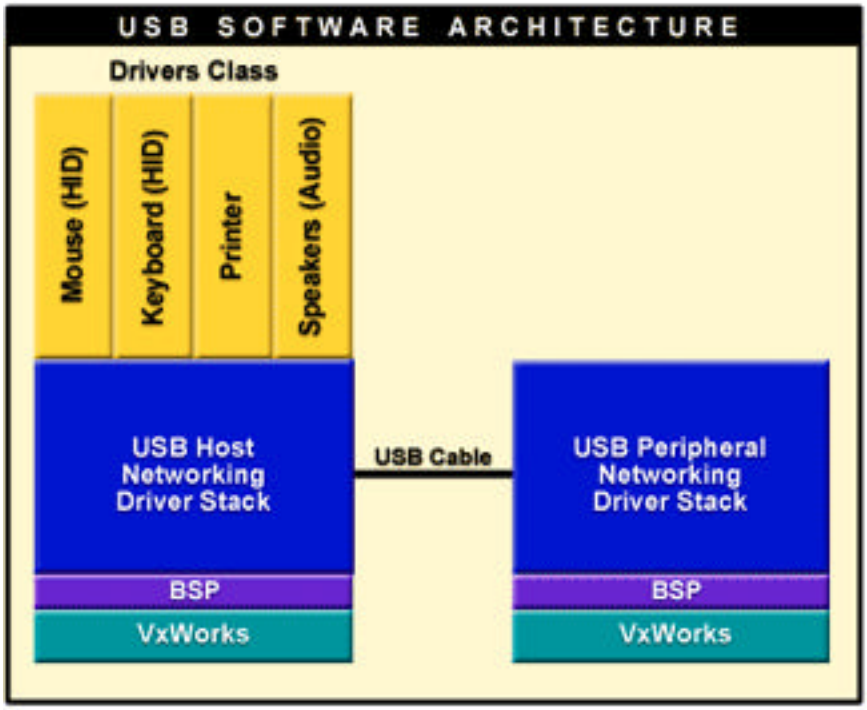

Figure 16-6. VxWorks supports USB Hosts and Clients

Components provided in source code

Wind River's USB Developer's Kit comes standard as a source-binary-source product. This means that the class drivers above the USB host driver stack, as well as the small layer of BSP-specific code below that stack, are provided in source code. The peripheral stack also comes in source code. The host driver stack comes standard in binary format, but source code for that stack is also available. This Developer's Kit is included on the CDROM in the Chapter 16/ VxWorks directory.

The host stack supports both the universal host controller interface (UHCI) and the open host controller interface (OHCI). A USB host driver stack automatically identifies and enumerates peripherals attached to the bus, allocates bandwidth to those peripherals, then handles data flow between peripherals and their host device. Because these peripherals are hot-swappable, they can be connected or removed at any time.

Host class drivers supported by the USB Developer's Kit enable developers to connect the most common peripherals "out of the box." Class drivers include keyboard and mouse human interface devices (HIDs) as well as printer and speakers (audio). Class drivers also provide all forms of data transfer.

The USB peripheral driver stack is a slave-type driver stack that responds to requests from the host. The peripheral stack's main function is to pass data between customer-specific peripheral class drivers and their host. The peripheral stack can, however, initiate wake-up commands to the host after the power management software has put the USB device "to sleep" that is, put the network on standby in order to save battery power.

The host driver stack is implemented as a set of hierarchical modules that includes a USB Exerciser Utility as shown in Figure 16-7. **usbTool** is a test application that gives you interactive control of the USB host driver stack. The **usbTool** utility exports a single entry point, ***usbTool*()**, that invokes a command-line-driven interactive environment in which the operator can initialize components of the USB driver stack, interrogate each USB connected to the system, send USB commands to individual USB devices, and test other elements of the USB stack. The **usbTool** test application is most useful during development and testing and does not need to be included in a shipped product.

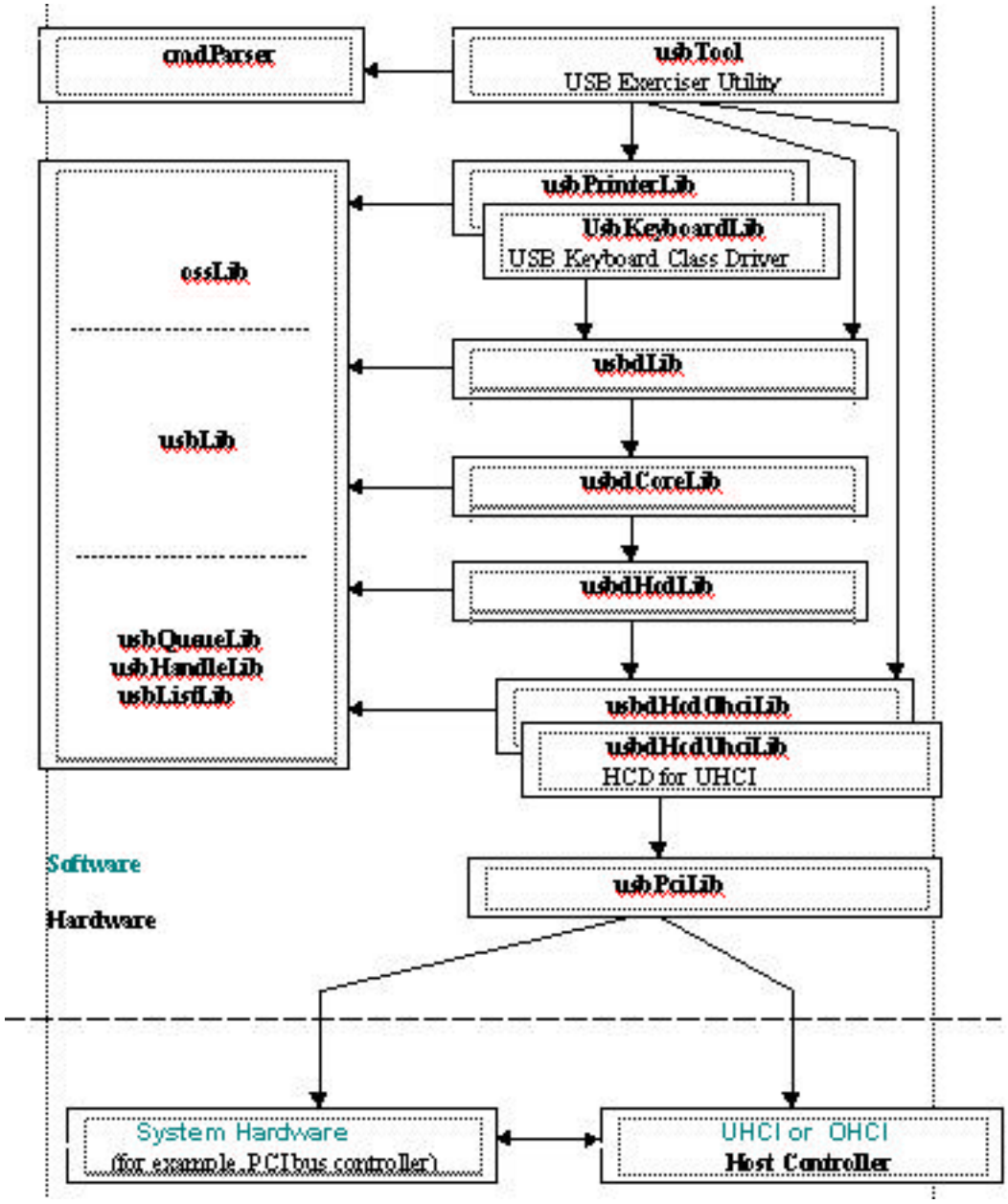

Figure 16-7. VxWorks' Host Controller Stack

Developing a client driver for an I/O device under VxWorks

The development of a client driver for a simple I/O device like “Buttons and Lights”, described in Chapter 7, will help a developer to understand how to implement USB under VxWorks.

The “Buttons and lights” device classifies itself under USB Human Interface Devices Class. By virtue of its general characteristics, this particular device can be treated as a simple I/O device under VxWorks. To begin with, the client driver has to be initialized so that it gets registered with USBD and also a call back has to be registered for Dynamic Attach Notification. This is boilerplate code for all USB devices and is shown in Figure 16-8.

Once the USB I/O device is attached it is configured as a new device with the newly created device structure on dynamic attachment. This too is boilerplate code used for any HID device and is included in Figure 16-8.

Once the IO request packect(IRP) is submitted, when ever there is data on the input pipe, the usbButtonsLightsIrpCallback() will be invoked. We need to read the data from the packet and set the lights using a SET_REPORT command. The IRP is initialized again and re-submitted. This completes the loop so that the input is continuously taken and the lights are set as per the switch settings.

Figure 16-8 includes segments the code to support our buttons and Lights HID example (the full source code and generation scripts are included on the CDROM). Note that our “application specific” code, that is, copying the buttons setting to the lights, is only three lines of Figure 16-8.

```
/*******************************************************************************
* notifyAttach - Notifies registered callers of attachment/removal
*/
        LOCAL VOID notifyAttach (pUSB_BTNLTS_SIO_CHAN pSioChan, UINT16 attachCode) {
        pATTACH_REQUEST pRequest = usbListFirst (&reqList);

while (pRequest != NULL)
                {
                (*pRequest->callback) (pRequest->callbackArg,
                (SIO_CHAN *) pSioChan, attachCode);
                pRequest = usbListNext (&pRequest->reqLink);
                }
        }
/*******************************************************************************
* initBtnltsIrp - Initialize IRP to listen for input on interrupt pipe
* RETURNS: TRUE if able to submit IRP successfully, else FALSE
*/
LOCAL BOOL initBtnltsIrp (pUSB_BTNLTS_SIO_CHAN pSioChan) {
        pUSB_IRP pIrp = &pSioChan->irp;
/* Initialize IRP */
        memset (pIrp, 0, sizeof (*pIrp));
        pIrp->userPtr = pSioChan;
        pIrp->irpLen = sizeof (*pIrp);
```

```
        pIrp->userCallback = usbButtonsLightsIrpCallback;
        pIrp->timeout = USB_TIMEOUT_NONE;
        pIrp->transferLen = 1;
        pIrp->bfrCount = 1;
        pIrp->bfrList [0].pid = USB_PID_IN;
        pIrp->bfrList [0].pBfr = &(pSioChan->btn_statusReq);
        pIrp->bfrList [0].bfrLen = 1;
/* Submit IRP */
        if (usbdTransfer (usbdHandle, pSioChan->pipeHandle, pIrp) != OK)
            return FALSE;
        pSioChan->irpInUse = TRUE;
        return TRUE;
        }
/*******************************************************************************
* usbButtonsLightsIrpCallback - Invoked upon IRP completion/cancellation
* Examines the cause of the IRP completion.  If completion was successful,
*  interprets the USB buttonslights's boot report and re-submits the IRP.
*/
LOCAL VOID usbButtonsLightsIrpCallback (pVOID p   /* completed IRP */ ) {
        static UINT8 ledReport = 0xff;
        UINT8 pre_ledStatus;
        pUSB_IRP pIrp = (pUSB_IRP) p;
        pUSB_BTNLTS_SIO_CHAN pSioChan = pIrp->userPtr;
        OSS_MUTEX_TAKE (btnltsMutex, OSS_BLOCK);
/* Was the IRP successful? */
        if (pIrp->result == OK)
        {
/* Interpret the buttonslights report */
            pre_ledStatus =   ledReport;
            ledReport = pIrp->bfrList[0].pBfr[0];
            if(pre_ledStatus != ledReport)
                printf(" LED Status : %d \n", ledReport);
            usbHidReportSet (usbdHandle, pSioChan->nodeId, pSioChan->interface,
            USB_HID_RPT_TYPE_OUTPUT, 0, &ledReport, sizeof (ledReport));
            }
/* Re-submit the IRP unless it was canceled - which would happen only
    during pipe shutdown (e.g., the disappearance of the device). */
        pSioChan->irpInUse = FALSE;
        if (pIrp->result != S_usbHcdLib_IRP_CANCELED)
            initBtnltsIrp (pSioChan);
        OSS_MUTEX_RELEASE (btnltsMutex);
        }
```

Figure 16-8. Buttons-and-Lights under VxWorks

EEPROM LOADER

One problem that we had to solve while debugging this example was loading the firmware into the EZUSB-based Buttons and Lights board. We did not have time to write an auto-loader for VxWorks (but check the web site, it may be completed by the time that you are reading this) so we resorted to an EEPROM loader. The USBSIMM board contains an 8KB I2C EEPROM and, if the EEPROM has a special key in byte 0, then the EZUSB component will load it's program from this EEPROM rather than across USB. Since there are many other

occasions that this capability would be useful I decided to solve this will a self-copying program.
The EEPROM must have a seven byte header followed by load modules. A load module is a <load address word><load length word><data, length bytes>. The load length cannot be greater than 1K (3FFH) so large programs must be broken into smaller load modules. The last load module must have bit 16 set of the load length and must be a byte write of 0 to memory location 7F92H so that the 8051 will be brought out of RESET.
I created a PROGEE.A51 which wraps itself around the example program such that it becomes a load module. Fortunately, the Buttons and Lights program uses less than 1KB. If you develop programs that are greated than 1K then you should break them into smaller blocks by adding four byte headers at strategic places throughout your code. PROGEE.A51 then makes itself the MAIN module so that it will run when the program is loaded on the USBSIMM board. Click GO to copy the example program, in the correct format, into the EEPROM. The source of PROGEE.A51 is included on the CDROM.

CHAPTER SUMMARY

We looked at two alternate operating systems, Linux and the VxWorks RTOS and discovered that USB is well supported. Building a simple HID application such as our Buttons and Lights example was very easy due to the underlying support provided by these operating systems. Other class drivers, such as hub, audio, printer and mass storage are also supported so many of the other examples in this book could also be easily ported.
There are a few implementation differences of course, but the partitioning of device drivers into class drivers plus miniport drivers is common throughout these, and others, USB-aware operating systems making it a straightforward process to move a USB product amongst different platforms. Indeed, it is now commonplace to use a USB-connected hard disk to move files between Windows, Linux, Windows CE and the Mac OS. The communications on the USB cable is protocol based and is independent of the PC host OS and the I/O device implementation. It is this freedom that has helped make USB one of the fastest growing interfaces.

I hope that I have helped you get up to speed so that you too can take advantage of the growing opportunities around USB.